

Partial mutual exclusion for infinitely many processes

Wim H. Hesselink, December 7, 2011
 Dept. of Computing Science, University of Groningen
 P.O.Box 407, 9700 AK Groningen, The Netherlands
 Email: w.h.hesselink@rug.nl

Abstract. Partial mutual exclusion is the drinking philosophers problem for complete graphs. It is the problem that a process may enter a critical section CS of its code only when some finite set nbh of other processes are not in their critical sections. For each execution of CS , the set nbh can be given by the environment. We present a starvation free solution of this problem in a setting with infinitely many processes, each with finite memory, that communicate by asynchronous messages. The solution has the property of first-come first-served, in so far as this can be guaranteed by asynchronous messages. For every execution of CS and every process in nbh , between three and six messages are needed. The correctness of the solution is argued with invariants and temporal logic. It has been verified with the proof assistant PVS.

Key words: Drinking philosophers; distributed algorithms; starvation freedom; verification; fairness

1 Introduction

Partial mutual exclusion is a problem that goes back to Dijkstra’s dining philosophers [5] and the drinking philosophers of Chandi and Misra [3]. It is the problem that a process may enter a critical section of its code only when some specified finite set nbh of other processes (neighbours) are not in their critical sections. In the case of the dining philosophers, the philosophers form a ring and nbh consists of the two neighbour philosophers.

The drinking philosophers form an arbitrary finite undirected graph, say with $Nbh.p$ as set of neighbours of philosopher p . The set $nbh.p$ is then a modifiable subset of $Nbh.p$, but the algorithm has a message complexity proportional to the sizes of $Nbh.p$, and these can be considerably larger than $nbh.p$.

Investigations into dining or drinking philosophers have always been motivated by the desire for a clean abstraction of the problems of resource allocation.

Inspired by the emergence of the internet, we generalize the setting to potentially infinitely many processes and to arbitrary finite sets nbh that can change over time. More precisely, when the environment prompts a process, say p , to enter its critical section CS , it gives p nondeterministically a finite set $nbh.p$ of other processes (to be called neighbours). After executing CS , process p resets $nbh.p$ to the empty set when it becomes idle again.

Partial mutual exclusion is the requirement that when two processes, say q and r , are both in each other’s neighbourhoods, they are not both in their critical sections:

$$PMX: \quad r \in nbh.q \wedge q \in nbh.r \wedge q \text{ in } CS \wedge r \text{ in } CS \Rightarrow \text{false} .$$

We do not require that $r \in nbh.q$ be equivalent to $q \in nbh.r$, because this would restrict the environment and the processes considerably. When q and r are both in each other’s neighbourhoods, we speak of a *conflict* between q and r . For comparison, mutual exclusion itself would be the requirement:

$$q \text{ in } CS \wedge r \text{ in } CS \Rightarrow q = r .$$

We assume that the processes have private memory and communicate by asynchronous messages. Messages are reliable: they are not lost or duplicated. They can pass each other, however. The processes receive and answer messages even when they are idle.

We impose a *first-come first-served* order (FCFS) in the following way. Whenever a process starts the entry protocol, it notifies all its neighbours. If the notification of process q reaches r before process r starts its entry protocol, and if the entry of r is in conflict with q , then process q reaches *CS* before process r does. The classical definition of FCFS in [10] only applies to shared memory systems and total mutual exclusion. The above definition of FCFS seems to be the natural translation for partial mutual exclusion with message passing.

As a process may have to wait a long time before it can enter the critical section, it is important that the environment of any process p is allowed nondeterministically to abort the entry protocol of p and move it back to the idle state.

There are two progress requirements: starvation freedom and maximal concurrency. *Starvation freedom* ([6], also called *lockout freedom* [13]) means that, for every p , whenever process p needs to enter *CS*, it will eventually do so if it is not aborted.

Starvation freedom can only hold under the assumption of weak fairness for all processes. Weak fairness for process p means that if, from some time onward, process p is continuously enabled to do a certain step different from entering or aborting, it will do the step. We elaborate on the concept of weak fairness in Section 4.

The second progress requirement is *maximal concurrency*, introduced in [15]. Maximal concurrency means that in any case (i.e. without weak fairness) a process can make progress when it has no conflicts with other processes. More precisely, every process p that needs to enter *CS* and does not abort, will eventually enter *CS*, provided it satisfies weak fairness itself, all other processes receive and answer messages from p , and no process comes in an eternal conflict with p . The point here is that processes without conflicts with p need not be weakly fair; e.g., they are allowed to remain in *CS* forever.

1.1 Sketch of a solution

The main reason for waiting is formed by *conflicts*: $q \in nbh.r$ and $r \in nbh.q$. The conflict relation determines an undirected graph. The performance of any solution depends on the sizes and the shapes of the connected components of this graph. Conversely, however, the evolution of this graph depends on the performance. The sooner a process reaches *CS*, the sooner its set *nbh* is made empty again. This justifies the search for a simple solution with as few messages as possible.

The first step of the solution was explicitly writing down what FCFS would mean for partial mutual exclusion in a message passing system. This led us to a solution with two layers: an outer protocol to guarantee FCFS, and an inner protocol to guarantee partial mutual exclusion. This is the same global design as used in the shared-variable mutual exclusion algorithm of [12].

The outer protocol requires 3 messages for every element of *nbh*. The inner protocol is asymmetric. We represent the processes by natural numbers. If q and r are processes with $q < r$, we speak of q as the lower process and r as the higher process. The inner protocol for process p requires 3 messages for every higher process in *nbh.p*, and no messages for the lower processes in *nbh.p*.

1.2 Restrictions

Although we allow infinitely many processes, we restrict the number of messages and the memory requirements of the processes in realistic ways.

For the correctness of the algorithm, the time needed for message transfer can be unbounded. For other issues, however, it is convenient to postulate an upper bound Δ for the time needed to execute an atomic command plus the time needed for the messages sent in this command to be received.

We define the *extended neighbourhood* of process p to be the union of $nbh.p$ with its dual $nbh^0.p = \{q \mid p \in nbh.q\}$. The k th delayed extended neighbourhood of p consists of the processes that were in p 's extended neighbourhood not longer than a time $k\Delta$ ago.

As there are many processes, we impose the communication restriction (CR) that every process only sends messages to the members of some delayed extended neighbourhood, and the memory restriction (MR) that every process only remembers relationships with members of some delayed extended neighbourhood.

It is fairly easy to see that our solution has these properties, using the first delayed extended neighbourhood for (CR) and the second one for (MR). We do not verify it formally, because this is cumbersome and not illuminating.

1.3 Overview and verification

We briefly discuss related research in Section 1.4. In Section 1.5, we describe the modelling of the asynchronous messages. Section 2 describes the algorithm. In Section 3, we prove its safety properties mutual exclusion and deadlock freedom. In Section 4, we prove the liveness properties starvation freedom and maximal concurrency.

The proofs of the safety and liveness properties have been carried out with the interactive proof assistant PVS [14]. The descriptions of proofs closely follow our PVS proof scripts, which can be found on our web site [7]. It is our intention that the paper can be read independently, but the proofs require so many case distinctions that manual verification is problematic. Section 5 provides a brief sketch of how we use PVS. We conclude in Section 6.

1.4 Related research

We solve the drinking philosophers' problem for an infinite complete graph, as a clean idealization of the resource allocation problem. We do not intend to solve the general resource allocation problem itself, as formulated in [17,2,15].

In the drinking philosophers' problems of [3,17] (also [13, Section 20]), the philosophers form a fixed finite undirected graph. Our set $nbh.p$ is then a subset of the constant set $Nbh.p$ of p 's neighbours in the graph. This subset is chosen by the environment each time that the process gets thirsty. The message complexity of the solutions of these papers is proportional to the size of $Nbh.p$ and not to the possibly considerably smaller size of $nbh.p$. These solutions are therefore problematic for large complete graphs. To enforce starvation freedom, these solutions assign directions to the edges of the graph such that the resulting digraph is acyclic.

We are not aware of other solutions to the partial mutual exclusion problem as formulated here.

The papers [17,15] offer a modular approach to the general resource allocation problem. This modular approach seems to correspond to the outer protocol in our solution, but in either case, the code is much more complicated than our outer protocol. The solution of [2] satisfies FCFS, but it is more complicated and needs much more messages than our solution.

1.5 Messages and channels

As announced, we assume that the processes communicate by asynchronous messages. Messages are guaranteed to arrive and be received, but the delay is unknown,

and messages are allowed to pass each other, unlike in [2,13] where the messages in transit, say from q to r are treated first-in-first-out.

Formally, we write $m.q.r$ to denote the number of messages m sent by q to r . In particular, it is a natural number. Sending corresponds to the incrementation $m.q.r++$. Receiving corresponds to decrementation $m.q.r--$ and has the precondition $m.q.r > 0$. In principle, $m.q.r$ can be any natural number, but in our algorithm, we take care to preserve the invariants $m.q.r \leq 1$: there is always at most one message m in transit from q to r . Initially, no messages are in transit: $m.q.r = 0$ for every message type m .

In CSP [8], one would write $m.q.r!$ for sending and $m.q.r?$ for receiving, but messages in CSP are synchronous. In Promela, the language of the model checker Spin [9], one could model the messages by channels with buffer size 1.

The view of the messages sent through channels must be taken with a grain of salt, because it stretches the imagination to declare for every process infinitely many channels. For implementation purposes, we prefer to regard a message m sent by q to r as a tuple (m, q, r) . Every process searches the tuple space continuously or repeatedly for messages with itself as destination.

2 The Algorithm

The code for each process is decomposed as a parallel composition of three component processes: an environment, a forward stepping component, and a component that receives the messages:

process(p) :
environ(p) || **forward**(p) || (|| q : **receive**(q, p)) .

The component **environ**(p) implements the environment's decisions for p : to start the entry protocol when it is idle, or to abort the entry protocol if needful. In component **forward**(p), process p traverses the entry protocol towards CS , followed by the exit protocol back to the idle state. For every q , component **receive**(q, p) serves to receive and handle all messages from q to p .

Each component of process p is an infinite loop, the body of which is a non-deterministic choice between several guarded alternatives as in Unity [4]. The guard of an alternative is its enabling condition. In **environ**(p) and **forward**(p), this is primarily the value of the program counter pc of the process (called *line number*). In most cases of **receive**(q, p), the presence of a message is the guard for its reception, and the first action is the removal of the message.

An important difference between **forward**(p) and **receive**(q, p) as opposed to **environ**(p), is that the alternatives of **forward**(p) and **receive**(q, p) are executed under weak fairness, i.e., if in some execution one of its alternatives is for some point onward continuously enabled this alternative will eventually be executed. On the other hand, the environment is never forced to act. We come back to this in Section 4.

One can argue that the critical section CS should be executed by the environment. We reckon it to **forward** instead, because the environment is allowed to do nothing, while CS needs to terminate.

2.1 A layered solution

If v is a private variable, outside the code, the value of v for process q is denoted by $v.q$. In the algorithm, every process has private variables *nbh*, *prio*, *before*, *after*, *wack*, *away*, *need*, *prom*, which all hold finite sets of processes, and which are

initially empty. Every process has a program counter $pc : \mathbb{N}$, which is initially 11. The role of the ghost variable *fork* is explained in Section 2.3.

We use five message types **req**, **gra**, **notify**, **withdraw**, and **ack**. As explained in Section 1.5, **req**. $q.r$ stands for the number of **req** messages from q to r , and analogously for the other message types.

The components of our solution are given in the Figures 1, 2, 3. We can regard the alternatives as atomic commands because actions on private variables give no interference, the messages are asynchronous, and any delay in sending a message can be regarded as a delay in message delivery.

```

environ( $p$ ) :
  ||  $pc = 11 \rightarrow \text{choose } nbh \text{ with } p \notin nbh ; pc := 12 .$ 
  ||  $pc = 12 \wedge p \in AE \rightarrow nbh := \emptyset ; pc := 11 .$ 
  ||  $pc = 13 \wedge p \in AE \rightarrow$ 
    for  $q \in nbh$  do withdraw. $p.q++$  od ;
     $wack := nbh ; nbh := \emptyset ; prio := \emptyset ; pc := 11 .$ 
  ||  $pc = 14 \wedge need \cap \{q \mid p < q\} = \emptyset \wedge p \in AE \rightarrow$ 
    for  $q \in nbh$  with  $p < q$  do
      gra. $p.q++$  ;  $fork.p.q--$  od ;
    for  $q \in nbh$  do withdraw. $p.q++$  od ;
     $wack := nbh ; need := \emptyset ; nbh := \emptyset ; pc := 11 .$ 

```

Fig. 1. The environment, to trigger and abort

A process p is called *idle* when $pc.p = 11$. When p is idle, the environment may decide to trigger the process by giving it a finite set $nbh.p$ with $p \notin nbh.p$, and setting $pc.p := 12$. See line 11 of **environ**. As $nbh.p$ will only be modified again when process p becomes idle again, we postulate the invariant

$$Iq0: \quad r \in nbh.q \Rightarrow q \neq r \wedge q \text{ in } \{12 \dots\} .$$

For a process q and a line number ℓ , we write q **at** ℓ to express $pc.q = \ell$. If L is a set of line numbers, we write q **in** L to express $pc.q \in L$. For all invariants, we implicitly universally quantify over the free variables, usually q and r .

At line 12, the process starts the entry protocol in the component **forward**. When waiting at lines 12, 13, or 14 takes too long, the environment may be allowed to abort **forward** and to go back to the idle state. The fixed unspecified set AE (abort enabled) is introduced to make it impossible that the aborting steps of the environment are used to prove progress properties (by accident or design).

As announced, the solution consists of two layers: an outer protocol for FCFS, and an inner protocol for mutual exclusion. The partition in layers is orthogonal to the partition in components. In **forward**(p), the outer protocol is visible in line 12, in the guard of line 13, and in the body of line 14. The inner protocol is visible in the body of line 13, in the guard of line 14, and in line 16 as a whole.

2.2 The outer protocol for FCFS

In order to guarantee FCFS, every process q maintains a set *before*. q of the processes that have sent notifications to q without withdrawing them. Indeed, when a process p has sent notifications, it needs to withdraw them when it enters *CS*, so that p does not force other processes needlessly to wait when it is in its exit protocol or idle again.

Because the messages are asynchronous and not necessarily FIFO, the message **withdraw** can arrive earlier than **notify**. We therefore treat the messages **notify**

```

forward( $p$ ) :
  ||  $pc = 12 \wedge wack = \emptyset \rightarrow$ 
    for  $q \in nbh$  do notify. $p.q++$  od ;
     $prio := nbh \cap (before \setminus after)$  ;  $pc := 13$  .
  ||  $pc = 13 \wedge prio = \emptyset \rightarrow$ 
    for  $q \in nbh$  with  $p < q$  do req. $p.q++$  od ;
     $need := \{q \in nbh \mid p < q \vee q \in away\}$  ;  $pc := 14$  .
  ||  $pc = 14 \wedge need = \emptyset \rightarrow$ 
    for  $q \in nbh$  do withdraw. $p.q++$  od ;
     $wack := nbh$  ;  $pc := 15$  .
  ||  $pc = 15 \rightarrow CS$  ;  $pc := 16$  .
  ||  $pc = 16 \rightarrow$ 
    for  $q \in nbh$  with  $p < q$  do gra. $p.q++$  ; fork. $p.q--$  od ;
     $nbh := \emptyset$  ;  $pc := 11$  .

```

Fig. 2. The stepping component

```

receive( $q, p$ ) :
  || notify. $q.p > 0 \rightarrow$  notify. $q.p--$  ; add  $q$  to  $before$  .
  || withdraw. $q.p > 0 \rightarrow$  withdraw. $q.p--$  ;
    remove  $q$  from  $prio$  ; add  $q$  to  $after$  .
  ||  $q \in after \cap before \rightarrow$ 
    remove  $q$  from  $after$  and  $before$  ; ack. $p.q++$  .
  || ack. $q.p > 0 \rightarrow$  ack. $q.p--$  ; remove  $q$  from  $wack$  .
  || req. $q.p > 0 \rightarrow$  req. $q.p--$  ; add  $q$  to  $prom$  .
  || gra. $q.p > 0 \rightarrow$  gra. $q.p--$  ; fork. $p.q++$  ;
    remove  $q$  from  $away$  and  $need$  .
  ||  $q \in prom \setminus away \wedge \neg (pc \geq 15 \wedge q \in nbh) \rightarrow$ 
    gra. $p.q++$  ; fork. $p.q--$  ;
    add  $q$  to  $away$  ; remove  $q$  from  $prom$  ;
    if  $pc = 14 \wedge q \in nbh$  then add  $q$  to  $need$  endif .

```

Fig. 3. The component of p receiving messages from q

and **withdraw** symmetrically. Arrival of **withdraw** is registered in *after*. When both have arrived, the combination is answered by a message **ack**, to preclude interference when **notify** or **withdraw** would be delayed. Each process holds in *wack* the set of processes it is expecting acknowledgements from. See the first four alternatives of **receive**(q, p). Note that idle processes also accept messages.

According to the definition of FCFS, when process p starts its entry protocol in **forward**(p), it sends messages **notify** to its neighbours, it forms a set $prio.p$ as the intersection of $nbh.p$ with the difference between $before.p$ and $after.p$, and then waits for the set $prio.p$ to become empty.

When process p enters *CS*, it withdraws all its outstanding notifications by sending **withdraw** messages to its neighbours, because at that point it cannot be overtaken anymore. It also sets $wack := nbh$. When it arrives again at line 12, it waits for *wack* to be empty. In this way, it verifies that all its messages **notify** and **withdraw** have been acknowledged.

The third alternative of **receive** is called **after** because the condition $q \in after.p$ is its usual trigger. This alternative can be eliminated by including it conditionally in both the first and the second alternative. We have not done so, because it would complicate the code. It may seem to be simpler to require separate acknowledgements for **notify** and **withdraw**. This, however, would require more messages and more waiting conditions.

In early drafts of the algorithm, we had located the waiting for the acks at line 16, as in Szymanski’s algorithm [16], but we have rotated it to line 12 to avoid unnecessary waiting.

The outer protocol thus uses the messages **notify**, **withdraw**, **ack**, and the private variables *prio*, *before*, *after*, and *wack*.

2.3 The inner protocol

The inner protocol serves to ensure partial mutual exclusion. It is inspired by the drinking philosophers of [3] but we do not insist on a symmetric solution. The idea is that the processes form a modifiable directed complete graph, which need not remain acyclic. For every edge, say between processes q and r , the direction is determined by a “fork” (or “bottle”) that is held either by q or by r , or that is in transit between them. We use an integer variable $fork.q.r$ (private to q) to count the number of forks to r that q holds.

In accordance with [5,3], we thus postulate the invariants

$$\begin{aligned} Iq1: & \quad q \neq r \Rightarrow fork.q.r + fork.r.q \leq 1, \\ Iq2: & \quad q \text{ in } \{15 \dots\} \wedge r \in nbh.q \Rightarrow fork.q.r > 0. \end{aligned}$$

As *CS* is in line 15, condition *PMX* is clearly implied by *Iq0*, *Iq1*, and *Iq2*. The values of $fork.q.q$ are irrelevant.

The basic idea of the inner protocol is that when process p needs to enter *CS*, it sends request messages **req** to some neighbours q for which it misses the fork, i.e., with $fork.p.q = 0$. When it has all forks needed, it can enter *CS*. When process p receives a request for a fork that it does not need, it grants it by sending a **gra** message.

At this point, we break the symmetry between processes, in two ways. Recall that we represent the processes by natural numbers, and that, if $q < r$, we say that process q is *lower* and that r is *higher*. We decide to give priority to the lower process. It follows that in the inner protocol, the lowest process waiting for forks can make progress.

In view of the memory restriction (MR) of Section 1.2, we cannot use the infinite array $fork.p$ as an actual private variable of p . We therefore treat $fork$ as a ghost variable, which can be eliminated from the algorithm but only serves in its description and its proof.

We eliminate the variable $fork$ from the algorithm by introducing private variables to express how $fork.p$ differs from a default fork distribution to be introduced next. As the processes are represented by natural numbers, there are two natural candidates for a default fork distribution. Because the lower process has priority and must therefore request the fork whenever it needs it, we locate the fork by default at the higher process. The *default fork distribution* therefore has $fork.q.r = |r < q|$, where we use $|b| = (b ? 1 : 0)$ for Boolean b . After every transaction this state of affairs is restored as much as possible.

Remark. It is possible to choose the alternative default fork distribution with $fork.q.r = |q < r|$. We come back to this in Section 6. \square

The set of forks missing from the default distribution for process p is registered in the private variable *away.p*. We thus postulate the invariant:

$$Iq3: \quad q \in away.r \equiv (q < r \wedge fork.r.q = 0).$$

Forks that are present despite the default fork distribution, are present because the process asked for them. They are recorded in the set difference $nbh \setminus need$.

In view of the default fork distribution, a process p that needs forks in line 13 sends requests **req** only to the higher neighbours q . If process q receives the request,

it puts p in its private variable $prom.q$ (promise). When $p \in prom.q$ and q has the fork, and is not currently using it, process q sends the fork by a message **gra**, and updates its administration by adapting *fork* and *away*. This last alternative of **receive** is called **prom**. It can be eliminated by including it conditionally in the alternatives **req** and **gra**, and in line 16.

When process p receives a fork by a **gra** message, it accepts the fork and updates the administration. At line 16, process p sends the forks it has used back to its higher neighbours in accordance to the default fork distribution.

To summarize, the inner protocol uses the messages **req** and **gra**, the private variables *away*, *need*, *prom*, and the ghost variables *fork*. Initially, the ghost variables *fork* satisfy the default fork distribution $fork.q.r = |r < q|$.

2.4 Informal correctness arguments

We postpone the full and formal proofs of correctness to Section 3 for safety, and to Section 4 for liveness. Here, we only give some indications.

The proof that the inner protocol guarantees partial mutual exclusion (*PMX*) is a matter of careful fork administration. This is relatively easy. The formal treatment is in Section 3.1.

Absence of deadlock is more complicated. There are three waiting conditions at the lines 12, 13, and 14 of **forward**, which each or in combination potentially could lead to deadlock. The waiting condition at line 12 is harmless, however, because it is just waiting for messages to arrive. The waiting condition at line 13 does not lead to deadlock, essentially because the process(es) waiting longest at line 13 can proceed to line 14. The waiting condition at line 14 does not lead to deadlock because the lowest process waiting at line 14 has priority. The formal treatment of deadlock is in Section 3.2.

Starvation freedom is the most complicated property. The inner protocol itself is not starvation free, because a lower process can claim priority over a higher process. Note, however, that when it does so, it will send the fork to the higher process in its exit protocol. The outer protocol is starvation free because it satisfies FCFS. When a process in the inner protocol is passed by another process, this other process cannot pass again because it will be blocked by the FCFS property of the outer protocol. As the number of processes in the inner protocol is finite, it follows that the combination of the protocols is starvation free. The formal treatment is in Section 4.

2.5 Message complexity and waiting times

In the outer protocol, process p exchanges 3 messages (**notify**, **withdraw**, **ack**) with every neighbour. In the inner protocol, it exchanges 3 messages (**req**, **gra**, **gra**) with every higher neighbour. In total it exchanges between 3 and 6 messages with every neighbour.

Component **forward** has 3 waiting conditions: emptiness of *prio* at line 13 to ensure FCFS, emptiness of *need* at line 14 to ensure mutual exclusion, and emptiness of *wack* at line 12 to preclude interference of delayed messages.

Recall from Section 1.2 that Δ serves as an upper bound of the time needed for the execution of an alternative, plus the time needed for reception of the messages sent. Therefore, waiting for emptiness of *wack* should not take more than 2Δ .

When the environment of p wants to abort the entry protocol at line 14, it may need to wait for emptiness of the higher part of *need*. This waiting is also short because process p has priority over its higher neighbours. If Γ is an upper bound for the execution time of *CS*, the higher part of *need* is empty after $\Gamma + 2\Delta$.

The important waiting conditions are therefore emptiness of *prio* at line 13 and emptiness of *need* at line 14. The first condition is unavoidable and completely determined by FCFS. The waiting time for emptiness of *need* at line 14 depends on the number of conflicting processes that are concurrently in the inner protocol. The outer protocol guarantees that conflicting processes do not enter the inner protocol concurrently unless they are activated by the environment within a period Δ . If the environment often activates several conflicting processes within periods Δ , our algorithm may have performance problems. It seems likely, however, that other algorithms would have the same problem.

3 Verification of Safety

In a distributed algorithm, at any moment, many processes are able to do a step that modifies the global state of the system. In our view, the only way to reason successfully about such a system is to analyse the properties that cannot be falsified by any step of the system. These are the invariants.

Formally, a predicate is called an *invariant* of an algorithm if it holds in all reachable states. A predicate J is called *inductive* if it holds initially and every step of the algorithm from a state that satisfies J results in a state that also satisfies J . Every inductive predicate is an invariant. Every predicate implied by an invariant is an invariant.

When a predicate is inductive, this is often easily verified. In many cases, the proof assistant PVS is able to do it without user intervention. It always requires a big case distinction, because the algorithm has 16 different alternatives in the Figures 1, 2, and 3.

Most invariants, however, are not inductive. Preservation of such a predicate by some alternatives needs the validity of other invariants in the precondition. We use PVS to pin down the problematic alternatives, but human intelligence is needed to determine the useful other invariants.

In proofs of invariants, we therefore use the phrase “*preservation of J at $\ell_1 \dots \ell_m$ follows from $J_1 \dots J_n$* ” to express that every step of the algorithm with precondition $J \wedge J_1 \dots J_n$ has the postcondition J , and that the additional predicates $J_1 \dots J_n$ are only needed for the alternatives $\ell_1 \dots \ell_m$. We indicate the alternatives of Figure 1 by **env11**, **env12**, **env13**, **env14**. The alternatives of Figure 2 are indicated by the line numbers. The alternatives of Figure 3 in which messages are received, are indicated by the message names **notify**, **withdraw**, **ack**, **req**, **gra**. The alternatives 3 and 7 of **receive** are indicated by **after** and **prom**, respectively.

The *follows from* relation makes the list of invariants into a directed graph. In our enumerations of invariants, we traverse this graph by breadth first search.

For all invariants postulated, the easy proof that they hold initially is left to the reader. We use the term invariant in a premature way. It will be justified at the end of the section.

Section 3.1 contains the proof that the algorithm satisfies the invariant *PMX* of partial mutual exclusion. Section 3.2 contains the proof of absence of deadlock. This proof uses invariants that are verified in Section 3.3.

3.1 The proof of mutual exclusion

In Section 2.3, we saw that the mutual exclusion predicate *PMX* is implied by *Iq0*, *Iq1*, and *Iq2*. This section contains the proof that *Iq0*, *Iq1*, *Iq2*, and *Iq3* of Section 2.3 are invariants. Firstly, it is easy to verify that *Iq0* is inductive. Predicate *Iq1* is implied by the observation that there is precisely one fork on every edge, as expressed by the invariant:

Iq4: $q \neq r \Rightarrow \text{fork}.q.r + \text{fork}.r.q + \text{gra}.q.r + \text{gra}.r.q = 1$.

Indeed, *Iq4* implies *Iq1* because **gra** holds natural numbers.

Predicate *Iq2* is implied by the invariants:

Iq5: $q \text{ in } \{14 \dots\} \wedge r \in \text{nbh}.q \Rightarrow r \in \text{need}.q \vee \text{fork}.q.r > 0$,

Iq6: $r \in \text{need}.q \Rightarrow q \text{ at } 14 \wedge r \in \text{nbh}.q$.

The invariant *Iq3* is a matter of careful fork administration. Preservation of *Iq3* at 16, **gra**, and **prom** follows from *Iq2*, *Iq4*, and the new invariants

Iq7: $\text{fork}.q.r \geq 0$,

Iq8: $q \in \text{prom}.r \Rightarrow q < r$.

Note that *Iq7* is not superfluous, because, in the algorithm, we unconditionally decrement *fork.q.r* in the alternatives 16 and **prom**. On the other hand, we treat the message variables *m.q.r* as natural numbers, because they are only decremented when positive.

Predicate *Iq4* is inductive. Preservation of *Iq5* at 13 and **gra** follows from *Iq0*, *Iq3*, and *Iq7*. Predicate *Iq6* is inductive. Preservation of *Iq7* at 16, **prom**, and **env14** follows from *Iq2*, *Iq3*, and *Iq8*.

Preservation of *Iq8* at **req** follows from the new invariant

Iq9: $\text{req}.q.r > 0 \Rightarrow q < r$.

Preservation of *Iq9* at 14 follows from *Iq8*.

It now follows that the conjunction of the universal quantification of the “invariants” introduced above is inductive. Therefore, each of them is itself invariant. In particular, the mutual exclusion predicate *PMX* is invariant. This concludes the proof that the algorithm satisfies *PMX*.

3.2 Absence of deadlock

We define a state to be *silent* when no process can do a step of **forward** or **receive** of Figures 2, 3. We define a state to be *in deadlock* when it is silent and some processes are not idle (not at 11). Note that the environment need not be disabled. Absence of deadlock is a safety requirement which will follow from the liveness requirement of starvation freedom. It is useful to prove absence of deadlock first, however, because the ingredients of this proof are bound to enter again in the more complicated proof of liveness.

For the proof of absence of deadlock, we need several additional invariants. Firstly, all processes are at the line numbers of the program (otherwise there are no steps). This amounts to the inductive invariant:

Jq0: $q \text{ in } \{11 \dots 16\}$.

Fork requests by the lower process are remembered as promises:

Jq1: $q < r \wedge r \in \text{need}.q \wedge \text{req}.q.r = \text{gra}.r.q = 0 \Rightarrow q \in \text{prom}.r$.

Predicate *Jq1* is invariant, because preservation at **prom** follows from *Iq8*.

The following invariants are more difficult. We only claim them here, and postpone the proofs to the next section.

Any process waiting for a fork, does not have it:

Jq2: $r \in \text{need}.q \Rightarrow \text{fork}.q.r = 0$.

Because of the default fork distribution, a lower process has and gets no fork unless it needs one:

Jq3: $q < r \wedge \text{fork}.q.r + \text{gra}.r.q > 0 \Rightarrow q \text{ in } \{14 \dots\} \wedge r \in \text{nbh}.q$.

When process q has sent **withdraw** to r , it expects an acknowledgement from r . It remembers this by putting r in its set *wack*. This is expressed in the inductive invariant:

Kq0: $\text{withdraw}.q.r + \text{ack}.r.q + |q \in \text{after}.r| = |r \in \text{wack}.q|$.

When q is in *after*. r and not in *before*. q , a **notify** message is in transit from q to r :

Kq1: $q \in \text{after}.r \Rightarrow \text{notify}.q.r > 0 \vee q \in \text{before}.r$.

A binary relation R is called *well-founded*, if every nonempty set S has an R -minimal element, i.e., an element $q \in S$ such that, for all $q' \in S$, we have $(q', q) \notin R$.

We now claim the invariant that the relation $\text{Prio} = \{(q, r) \mid q \in \text{prio}.r\}$ on the processes is always well-founded:

Lq0: $\text{well-founded}(\text{Prio})$.

We also need:

Lq1: $q \in \text{prio}.r \wedge \text{withdraw}.q.r = 0 \Rightarrow q \text{ in } \{13 \dots\}$.

Under assumption of these invariants, we can prove:

Theorem 1. *The system is deadlock free.*

Proof. Assume that the state is silent. The idle processes are those at 11. Because of *Jq0*, the nonidle processes are at line 12 waiting for emptiness of *wack*, at line 13 waiting for emptiness of *prio*, or at line 14 waiting for emptiness of *need*. We have to prove that all processes are idle.

Because the state is silent, there are no messages **req**, **gra**, **notify**, **withdraw**, and **ack**. By *Iq4*, it follows that $\text{fork}.r.q = 1 - \text{fork}.q.r$ for all pairs $q \neq r$. By *Kq1*, the set *after*. r is a subset of *before*. r for all r . As the alternative *after* is disabled for all r , it follows that *after*. r is empty for all processes r . By *Kq0*, it follows that *wack*. q is empty for all processes q . In particular, no process is disabled at line 12, and all processes are at the lines 11, 13, and 14.

First assume that there are processes at 14. Let p be the lowest process at 14. As p is disabled at line 14, the set *need*. p is nonempty, say $q \in \text{need}.p$. By *Jq2*, we have $\text{fork}.p.q = 0$ and hence $\text{fork}.q.p = 1$. If $q < p$, then q is at 11 or 13, contradicting *Jq3*. Therefore $p < q$ by *Iq6* and *Iq0*. As there are no **req** or **gra** messages in transit, it follows that $p \in \text{prom}.q$ by *Jq1*. By *Iq3*, we have $p \notin \text{away}.q$. As the alternative *prom* is disabled, it follows that $pc.q \geq 15$, a contradiction. This proves that there are no processes at 14.

We thus have that all disabled processes are at 13. Let S be the set of processes at 13. If S is empty, we are done. Therefore, assume that S is nonempty. By *Lq0*, the set S has a *Prio*-minimal element, say p . As p is disabled at 13, the set *prio*. p is nonempty, say $q \in \text{prio}.p$. As there are no **withdraw** messages, the invariant *Lq1* implies that q is in $\{13 \dots\}$. As all processes are at 11 or 13, this implies that $q \in S$ and $(q, p) \in \text{Prio}$, contradicting the minimality of p . Consequently, there are also no processes at 13. \square

3.3 Invariants against deadlock

In this section we prove the invariants *Jq2*, *Jq3*, *Kq0*, *Kq1*, *Lq0*, *Lq1*, postulated in the previous section. In the course of this proof we need to postulate and prove several other invariants. The reader may prefer not to verify these proofs, because we obtained and verified them interactively with the proof assistant PVS. What is

relevant is to see the kind of assertions that can be made and their logical relationships.

Preservation of $Jq2$ at 13, 16 and **prom** follows from $Iq2$, $Iq3$, $Iq4$, $Iq7$, $Iq8$, and $Jq3$. Preservation of $Jq3$ follows at 16, **prom**, **env14** from $Iq4$, $Iq6$, $Iq7$, and the invariant

$$Jq4: \quad q \in \text{prom}.r \Rightarrow r \in \text{need}.q \wedge \text{req}.q.r = \text{gra}.r.q = 0 .$$

Preservation of $Jq4$ at 13, 16, **req**, **env14** follows from $Iq6$, $Iq8$, $Jq5$ and the invariants

$$Jq5: \quad \text{req}.q.r > 0 \Rightarrow r \in \text{need}.q \wedge \text{gra}.r.q = 0 ,$$

$$Jq6: \quad \text{req}.q.r \leq 1 .$$

Preservation of $Jq5$ at 13, 16, **prom**, **env14** follows from $Iq6$, $Iq7$, $Iq9$, $Jq3$, and $Jq4$. Preservation of $Jq6$ at 13 follows from $Iq6$ and $Jq5$. Note that $Jq4$ together with $Iq8$ imply that the implication in $Jq1$ can be replaced by an equivalence.

Preservation of $Kq0$ at 14, **env13**, **env14** follows from the inductive invariant

$$Kq2: \quad q \text{ in } \{13, 14\} \Rightarrow \text{wack}.q = \emptyset .$$

Preservation of $Kq1$ at **withdraw** follows from the invariant

$$Kq3: \quad \text{withdraw}.q.r > 0 \wedge \text{notify}.q.r = 0 \Rightarrow q \in \text{before}.r .$$

Preservation of $Kq3$ at 14, **env13**, **env14**, and **after** follows from $Kq0$ and the new invariant

$$Kq4: \quad q \text{ in } \{13, 14\} \wedge r \in \text{nbh}.q \wedge \text{notify}.q.r = 0 \Rightarrow q \in \text{before}.r .$$

Preservation of $Kq4$ at **after** follows from $Kq0$ and $Kq2$.

In order to prove $Lq1$, we postulate the new invariant

$$Kq5: \quad q \in \text{before}.r \setminus \text{after}.r \wedge \text{withdraw}.q.r = 0 \\ \Rightarrow q \text{ in } \{13, 14\} \wedge r \in \text{nbh}.q .$$

Preservation of $Kq5$ at **notify** follows from the new invariant:

$$Kq6: \quad \text{notify}.q.r > 0 \wedge q \notin \text{after}.r \wedge \text{withdraw}.q.r = 0 \\ \Rightarrow q \text{ in } \{13, 14\} \wedge r \in \text{nbh}.q .$$

Preservation of $Kq6$ at 14, **env13**, **env14**, **after** follows from $Kq0$ and the new invariant

$$Kq7: \quad q \in \text{before}.r \Rightarrow \text{notify}.q.r = 0 .$$

Preservation of $Kq7$ at 12 and **notify** follows from $Kq0$, $Kq5$, and the new invariant:

$$Kq8: \quad \text{notify}.q.r \leq 1 .$$

Preservation of $Kq8$ at 12 follows from $Kq0$ and $Kq6$.

We turn to preservation of $Lq0$. Recall that, for a relation R and a set S , an element s is called R -minimal in S iff it satisfies $s \in S$, and $(s', s) \notin R$ for all elements $s' \in S$. Relation R is called well-founded iff every nonempty set S has an R -minimal element. $Lq0$ asserts that the relation $Prio$, which consists of the pairs (q, r) with $q \in \text{prio}.r$, is well-founded.

It is easy to verify that, if R is a well-founded relation, every subrelation $R' \subseteq R$ is also well-founded. Therefore, $Lq0$ is preserved by the modifications of prio in **withdraw** and **env13**, as these remove elements from $Prio$.

Preservation of $Lq0$ at 12 follows from $Iq0$, $Kq0$, and $Lq1$. This is proved as follows. Assume that process p executes line 12. Let us use $Prio$ for relation $Prio$ in the precondition, and $Prio^+$ for relation $Prio$ in the postcondition of this step. Assume that $Lq0$ holds in the precondition. Therefore, relation $Prio$ is well-founded. It is easy to see that

$$(0) \quad \text{Prio}^+ \subseteq \text{Prio} \cup \{(q, p) \mid q \in \text{nbh}.p\} .$$

Let S be a nonempty set of processes. First, assume that $S' = S \setminus \{p\}$ is nonempty. Therefore, S' has a *Prio*-minimal element $r \in S'$. Because p executes line 12, the set $\text{wack}.p$ is empty. Therefore $Kq0$ implies $\text{withdraw}.p.r = 0$, and $Lq1$ implies $p \notin \text{prio}.r$. Therefore r is also *Prio*-minimal in S . It follows that r is Prio^+ -minimal in S because of $r \neq p$ and formula (0). It remains the case that $S \setminus \{p\}$ is empty. Then we have $S = \{p\}$. It follows that p is a *Prio*-minimal element of S and, hence, a Prio^+ -minimal element of S because of $Iq0$.

In either case, the set S contains a Prio^+ -minimal element. This proves that Prio^+ is well-founded. Therefore $Lq0$ is preserved by the step of line 12.

Predicate $Lq1$ is implied by the conjunction of $Kq5$ and the new invariant:

$$Lq2: \quad \text{prio}.q \subseteq \text{before}.q \setminus \text{after}.q .$$

Predicate $Lq2$ is inductive. This concludes the proofs of the invariants against deadlock.

Remarks. Apart from $Lq0$, all these invariants concern at most two processes. It is therefore possible to verify them by model checking. Such a system with two processes would not have too many states. The proofs of $Lq0$ and of absence of deadlock, however, do require theorem proving.

The invariants of the lists Iq^* , Jq^* relate to the inner protocol. The invariants of the lists Kq^* , Lq^* are exclusively related to the outer protocol. The only invariant for both protocols is $Iq0$.

4 Liveness

The aim of this section is to show that the algorithm satisfies two liveness properties: starvation freedom and maximal concurrency.

Starvation freedom means that every process that needs to enter *CS* and does not abort, will eventually enter *CS* and come back to the idle state. This can only be proved when all processes make progress under weak fairness.

Maximal concurrency means that, even without global weak fairness conditions, when process p itself makes progress under weak fairness and all processes answer messages from p , then whenever process p needs to enter *CS* and does not abort, it will eventually enter *CS* and come back to the idle state, unless it is, from some moment onward, eternally in conflict with some other process. The latter case is not deadlock, because the other process is not locked but only fails to do steps as it is not subject to fairness conditions.

Weak fairness is introduced in Section 4.1. We formalize executions as state sequences in Section 4.2. Section 4.3 contains the formal definitions of weak fairness and the statements of starvation freedom and maximal concurrency.

The proofs of these two results are distributed over several subsections. In Section 4.4, we introduce two more invariants and show that all message channels are infinitely often empty. In Section 4.5, we build the machinery to reduce the proof obligations to progress at the lines 12, 13, and 14. The Sections 4.6, 4.7, 4.8 treat these lines. Section 4.9 concludes the proofs of the two theorems.

4.1 Weak fairness

First, however, weak fairness needs an explanation. Roughly speaking, a system is called weakly fair if, whenever some process from some point onward always can do a step, it will do the step. Yet if a process is idle, it must not be forced to be interested in *CS*. Similarly, if a process is waiting a long time in the entry protocol,

we do not want it to be forced to abort the entry protocol. We therefore do not enforce the environment to do steps. We thus exclude the environment from the weak fairness conditions.

We impose the following weak-fairness conditions. For any process p , if from some time onward p can continuously do a **forward** step, it will do the **forward** step. If, from some time onward, it can continuously receive a message m from some process q , it will receive m from q .

Formally, we do not argue about the fairness of systems, but characterize the executions they can perform. Recall that an *execution* is an infinite sequence of states that starts in an initial state and for which every pair of subsequent states satisfies the step relation. An execution is called *weakly fair* iff, for every process p , whenever p can, from some state onward, always do a **forward** step or receive a message m from q , it will eventually do the step or receive message m from q .

We also need that, when one of the alternatives **after** or **prom** of **receive**(q, p) is from some time onward continuously enabled, this alternative is eventually taken. In the following, we therefore treat these alternatives as if they correspond to messages $m = \mathbf{after}$ or **prom** from q to p .

4.2 Formalization

We formalize the setting in a set-theoretic version of (linear time) temporal logic. Let X be the state space. We identify the set X^ω of the infinite sequences of states with the set of functions $\mathbb{N} \rightarrow X$. For a state sequence $xs \in X^\omega$ and $n \in \mathbb{N}$, we occasionally refer to $xs(n)$ as the state at time n . For a programming variable v , we write $xs(n).v$ for the value of v in state $xs(n)$.

For a subset $U \subseteq X$, we define $\llbracket U \rrbracket \subseteq X^\omega$ as the set of infinite sequences xs with $xs(0) \in U$. For a relation $A \subseteq X^2$, we define $\llbracket A \rrbracket_2 \subseteq X^\omega$ as the set of sequences xs with $(xs(0), xs(1)) \in A$.

For $xs \in X^\omega$ and $k \in \mathbb{N}$, we define the shifted sequence $D(k, xs)$ by $D(k, xs)(n) = xs(k + n)$. For a subset $P \subseteq X^\omega$ we define $\Box P$ (*always* P) and $\Diamond P$ (*eventually* P) as the subsets of X^ω given by

$$\begin{aligned} xs \in \Box P &\equiv (\forall k \in \mathbb{N} : D(k, xs) \in P) , \\ xs \in \Diamond P &\equiv (\exists k \in \mathbb{N} : D(k, xs) \in P) . \end{aligned}$$

We now apply this to the algorithm. We write $init \subseteq X$ for the set of initial states and $step \subseteq X^2$ for the step relation on X . Following [1], we use the convention that relation $step$ is reflexive (contains the identity relation). An *execution* is an infinite sequence of states that starts in an initial state and in which each subsequent pair of states is connected by a step. The set of executions of the algorithm is therefore

$$Ex = \llbracket init \rrbracket \cap \Box \llbracket step \rrbracket_2 .$$

If J is an invariant of the system, it holds in all states of every execution. We therefore have $Ex \subseteq \Box J$.

For our algorithm, the step relation $step \subseteq X^2$ is the union of the identity relation on X (because $step$ should be reflexive) with the relations $step(p)$ that consists of the state pairs (x, y) where y is a state obtained when process p does a step starting in x . The steps that process p can do are summarized in

$$step(p) = env(p) \cup fwd(p) \cup \bigcup_{q, m} rec(m, q, p) ,$$

where $env(p)$ consists of the steps of **environ**(p), $fwd(p)$ consists of the steps of **forward**(p), and $rec(m, q, p)$ consists of the steps where p receives message m from q in **receive**. Note that we take the union here over all processes q and all seven alternatives m of **receive** (including **after** and **prom**).

We define $(q \text{ at } k)$ to be the subset of X of the states in which process q is at line k . An execution in which process q is always eventually at NCS , is therefore an element of $\Box\Diamond\llbracket q \text{ at } 11 \rrbracket$. The aim is to prove that all executions we need to consider are elements of this set.

Remark. Note the difference between $\Box\Diamond\llbracket U \rrbracket$ and $\Diamond\Box\llbracket U \rrbracket$. In general, $\Box\Diamond\llbracket U \rrbracket$ is a bigger set (a weaker condition) than $\Diamond\Box\llbracket U \rrbracket$. The first set contains all sequences that are infinitely often in U , the second set contains the sequences that are from some point onward eternally in U . \square

4.3 Liveness under weak fairness

For a relation $R \subseteq X^2$, we define the *disabled* set $D(R) = \{x \mid \forall y : (x, y) \notin R\}$. Now *weak fairness* [11] for R is defined as the set of state sequences in which R is disabled infinitely often or is taken infinitely often:

$$wf(R) = \Box\Diamond\llbracket D(R) \rrbracket \cup \Box\Diamond\llbracket R \rrbracket_2 .$$

For a single process p , weak fairness for the steps of **forward**(p) is the property $wf(\text{fwd}(p))$.

Our algorithm needs the property that every message, say m in transit from q to r , is eventually received. The set $wf(\text{rec}(m, q, r))$ contains precisely the state sequences that satisfy this condition. This also applies to $m = \text{after}$ and prom .

For some purposes, we need the assumptions of weak fairness for the steps of a single process p and for all messages with p as destination or source. We thus define the set of *executions weakly fair for p* by

$$Wf(p) = Ex \cap wf(\text{fwd}(p)) \cap \bigcap_{q,m} (wf(\text{rec}(m, q, p)) \cap wf(\text{rec}(m, p, q))) .$$

The set of (globally) *weakly fair executions* is defined by messages, as captured in

$$WF = Ex \cap \bigcap_p wf(\text{fwd}(p)) \cap \bigcap_{p,q,m} wf(\text{rec}(m, q, p)) .$$

We can now formulate our two liveness results. Starvation freedom means that every process p in every weakly fair execution is always eventually back at NCS (i.e. at line 11). This is expressed by

Theorem 2. $WF \subseteq \Box\Diamond\llbracket p \text{ at } 11 \rrbracket$ for every process p .

Maximal concurrency means that every process p that needs to enter CS and does not abort, will eventually enter CS , provided it satisfies weak fairness itself, all other processes receive and answer messages from p , and no process comes in an eternal conflict with p . Let us define $p \bowtie q$ to mean that p and q are in conflict, i.e., $q \in nbh.p \wedge p \in nbh.q$. Then maximal concurrency is expressed by

Theorem 3. $Wf(p) \subseteq \Box\Diamond\llbracket p \text{ at } 11 \rrbracket \cup \bigcup_q \Diamond\Box\llbracket p \bowtie q \rrbracket$ for every process p .

The remainder of this section is devoted to the proofs of these two theorems. These proofs have a significant overlap. On the other hand, the proof of Theorem 2 has similarities with the proof of absence of deadlock (Theorem 1 in Section 3.2).

4.4 Empty channels

At this point, we postulate two additional invariants:

$$\begin{aligned} Mq0: & \quad \text{gra}.q.q = 0 , \\ Mq1: & \quad r \in \text{prio}.q \Rightarrow q \text{ at } 13 \wedge r \in nbh.q . \end{aligned}$$

Predicate $Mq0$ is preserved at 16 and **req** because of $Iq9$ and $Iq8$. Predicate $Mq1$ is inductive.

We now claim that $m.q.r \leq 1$ always holds for all five message types m and all processes q and r . For **req**, **notify**, **withdraw**, **ack**, this follows from $Jq6$, $Kq8$, and $Kq0$. For **gra**, it follows from $Iq4$, $Iq7$, and $Mq0$.

As every state in every execution satisfies all invariants, and the reception of message m from q by r decrements $m.q.r$, it follows that we have

$$(1) \quad Ex \cap wf(rec(m, q, r)) \subseteq \Box \Diamond \llbracket m.q.r = 0 \rrbracket .$$

In words, every message channel is infinitely often empty.

One can do similar assertions about the alternatives **after** and **prom**, but this is not useful.

4.5 Treating the loop

We use the *leads-to* relation between state predicates of [4]. A predicate U is said to *lead to* V if it is always the case that if U holds, then eventually V holds. This is formalized as follows. For subsets U and V of X , the set of state sequences in which U *leads to* V is defined by

$$LT(U, V) = \Box(\neg \llbracket U \rrbracket \cup \Diamond \llbracket V \rrbracket) .$$

Our specific algorithm is a simple loop with the property that, in any execution, if some process p does not get stuck at a line k , it will eventually proceed to line $k + 1$ or to 11.

We need to prove that a process reaches 11 from a combination of lines. We therefore define

$$toIdle(k, p) = LT(p \text{ in } \{k \dots\}, p \text{ at } 11) .$$

The relevance of this concept follows from the inclusion:

$$(2) \quad Ex \cap toIdle(12, p) \subseteq \Box \Diamond \llbracket p \text{ at } 11 \rrbracket .$$

This holds because, in any execution that belongs to $toIdle(12, p)$, if at some time p is not at 11, then p is in $\{12 \dots\}$ by $Jq0$, and therefore p will return to 11.

On the other hand, in any execution, process p is never in $\{17 \dots\}$ by $Jq0$. We therefore have

$$(3) \quad Ex \subseteq toIdle(17, p) .$$

The aim is thus to decrement the first argument of $toIdle$. This is done with the relation

$$(4) \quad Ex \cap toIdle(k + 1, p) \subseteq toIdle(k, p) \cup \Diamond \Box \llbracket p \text{ at } k \rrbracket .$$

This formula just means that, in any execution, a process p that is ever at line k , but from that time onward never at line 11 or in $\{k + 1 \dots\}$, needs to remain at k .

As a process is never disabled at the lines 15 or 16, weak fairness implies that it never stays there, i.e., we have

$$Ex \cap wf(fwd(p)) \cap \Diamond \Box \llbracket p \text{ at } k \rrbracket = \emptyset \text{ for } k = 15, 16.$$

Therefore, the formulas (3) and (4) imply that

$$(5) \quad Ex \cap wf(fwd(p)) \subseteq toIdle(15, p) .$$

It remains to eliminate the executions in $\Diamond \Box \llbracket p \text{ at } k \rrbracket$ for $k = 12, 13$, and 14. This is done in the next three sections.

4.6 Progress at line 12

Consider an execution $xs \in \Diamond \Box [p \text{ at } 12]$. From some time n_0 onward, process p is and remains at line 12. Therefore, weak fairness of $fwd(p)$ implies that it is infinitely often disabled. It is disabled iff the set $wack.p$ is nonempty. While p is at line 12, the finite set $wack.p$ can only become smaller. It therefore is eventually constant and nonempty. So, there is a process q such that, from time n_0 onward, $q \in wack.p$ always holds.

Assume that $ack.q.p > 0$ holds at some time $n_1 \geq n_0$. Then, by formula (1), there is a time $n \geq n_1$ such that $ack.q.p = 0$; therefore the **ack** message is received and $q \in wack.p$ is falsified. This proves that $ack.q.p = 0$ holds at any time $n \geq n_0$.

By formula (1), there is a time $n_1 \geq n_0$ such that $withdraw.p.q = 0$. By $Kq0$, we then have $p \in after.q$. This can only be falsified by the alternative **after**, which sends a message **ack.q.p**. We therefore have that $p \in after.q$ holds at all time $n \geq n_1$. By formula (1), there is a time $n_2 \geq n_1$ such that $notify.p.q = 0$ and hence $p \in before.q$ by $Kq1$. This can only be falsified by the alternative **after** which sends a message **ack.q.p**. We therefore have that $p \in before.q$ holds at all time $n \geq n_2$. Therefore, the alternative **after** is eternally enabled from time n_2 onward. By weak fairness it will be taken, thus falsifying $p \in before.q$. This is a contradiction.

We have derived this contradiction using weak fairness of $fwd(p)$, and of $rec(m, p, q)$ and $rec(m, q, p)$ for all q and some m . We therefore have proved that

$$(6) \quad Wf(p) \cap \Diamond \Box [p \text{ at } 12] = \emptyset .$$

4.7 Progress at line 13

We now want to exclude the possibility that in some execution some process is eventually always at line 13. For an execution xs and a time $n \in \mathbb{N}$, let $S(n, xs)$ be the set of processes that, in xs , from time n onward, is always at 13. The invariant $Lq0$ in the state $xs(n)$ implies:

$$(7) \quad \begin{aligned} xs \in Ex \ \wedge \ S(n, xs) \neq \emptyset \\ \Rightarrow \exists p \in S(n, xs) : S(n, xs) \cap xs(n).prio.p = \emptyset . \end{aligned}$$

Let $p \in S(n, xs)$. From time n onward, process p is and remains at line 13. By weak fairness of $fwd(p)$, process p it is infinitely often disabled. As it is at line 13, process p is disabled iff the set $prio.p$ is nonempty. While p is at line 13, the finite set $prio.p$ can only become smaller. It therefore is eventually constant and nonempty. So, there is a process q such that from time n onward, $q \in prio.p$ always holds. It follows that, from time n onward, process p does not receive **withdraw** from q . By weak fairness of $rec(\text{withdraw}, q, p)$, it follows that $withdraw.q.p = 0$ holds from time n onward. We thus have proved:

$$(8) \quad \begin{aligned} xs \in Ex \ \wedge \ p \in S(n, xs) \ \wedge \ xs \in Wf(p) \\ \Rightarrow \exists q : \forall i : q \in xs(n+i).prio.p \ \wedge \ xs(n+i).withdraw.q.p = 0 . \end{aligned}$$

At this point, we note that the invariants $Kq5$, $Lq2$, and $Mq1$ together imply:

$$(9) \quad q \in prio.p \ \wedge \ withdraw.q.p = 0 \Rightarrow q \text{ in } \{13, 14\} \ \wedge \ p \bowtie q .$$

Therefore, formula (8) implies

$$(10) \quad Wf(p) \cap \Diamond \Box [p \text{ at } 13] \subseteq \bigcup_q \Diamond \Box [p \bowtie q] .$$

For the sake of starvation freedom, we combine (7) and (8) to

$$(11) \quad WF \cap \Diamond \Box [r \text{ at } 13] \subseteq \bigcup_q \Diamond \Box [q \text{ at } 14] .$$

This formula is proved as follows. Let xs be in the lefthand set. Then there is a number n such that $S(n, xs)$ is nonempty. Formula (7) gives some process $p \in S(n, xs)$ with $prio.p$ disjoint from $S(n, xs)$. As $WF \subseteq Wf(p)$, the formulas (8) and (9) yield a process q that is and remains in $prio.p$ and that is and remains in $\{13, 14\}$. As $q \notin S(n, xs)$, it follows that q is eventually always at 14.

4.8 Progress at line 14

Let xs be an execution in $\Diamond \Box [p \text{ at } 14]$. Process p waits at line 14 for emptiness of $need$. This condition belongs to the inner protocol. The inner protocol in isolation, however, is not starvation free because it would allow a lower process repeatedly to claim priority over p by sending **reqs**. We need the FCFS property of the outer protocol to preclude this. Technically, the problem is that $need.p$ can grow at line 14 in the alternative **prom**.

Partly, in order to prove that eventually the truth value of $q \in need.p$ is constant, we construct a numeric state function $vf(q, p) \geq 0$ that, for $q \in nbh.p$, eventually never increases and therefore stabilizes to a constant value, and that it is only constant when the truth value of $q \in need.p$ is also constant. We construct vf as the weighted sum of three bit-valued state functions:

$$\begin{aligned} vf(q, p) &= vf0(q, p) + 2 \cdot vf1(q, p) + 4 \cdot vf2(q, p) \quad \text{where} \\ vf0(q, p) &= |q \in need.p|, \\ vf1(q, p) &= |fork.q.p + gra.p.q = 0 \wedge q < p|, \\ vf2(q, p) &= |q \text{ in } \{13 \dots\} \wedge p \in nbh.q \wedge p \notin prio.q|. \end{aligned}$$

Indeed, when process p is and remains at line 14, its neighbourhood $nbh.p$ is constant. For any $q \in nbh.p$, we have eventually $notify.p.q = 0$ by formula (1). This remains valid because p at 14 does not send **notify**. By $Kq4$, we therefore have eventually always $p \in before.q$.

We claim that, while p is at line 14 and $p \in before.q$ holds, $vf(q, p)$ never increases. This is proved as follows. At line 12, $vf2(q, p)$ does not increase because of $Kq0$ and $Kq2$. The same holds for **withdraw**. At line 16, $vf1(q, p)$ can be incremented, but this is compensated by decrementation of $vf2(q, p)$ because of $Mq1$. The difficult alternative is **prom**, because it can increment $vf0(q, p)$ by adding q to $need.p$. In that case, $Iq8$ implies $q < p$. Therefore, $vf1(q, p)$ is decremented because of $Iq3$, $Iq4$, and $Iq7$. This proves the claim.

It follows that, if process p is and remains at line 14, eventually $vf(q, p)$ becomes constant. When $vf(q, p)$ is constant, $q \in need.p$ is also constant because $q \in need.p$ holds if and only if $vf(q, p)$ is odd. This proves that, eventually, the truth value of $q \in need.p$ becomes constant.

As $need.p$ is always a subset of the finite set $nbh.p$, which is constant while p is at line 14, we can now conclude that eventually $need.p$ is constant. If this constant would be the empty set, process p would be eventually always enabled, and by weak fairness, process p would leave line 14. Therefore, there is some process q eventually always in $need.p$. We have $q \neq p$ because of $Iq0$ and $Iq6$. This proves

$$(12) \quad Wf(p) \cap \Diamond \Box [p \text{ at } 14] \subseteq \bigcup_{q \neq p} \Diamond \Box [q \in need.p].$$

We now distinguish the cases $q < p$ and $p < q$. For the first case, we claim:

$$(13) \quad q < p \Rightarrow Wf(p) \cap \Diamond \Box [q \in need.p] \subseteq \Diamond \Box [q \text{ in } \{14 \dots\} \wedge p \in nbh.q].$$

This is proved as follows. Firstly, $q \in need.p$ implies $fork.p.q = 0$ by $Jq2$. By (1) we have infinitely often $gra.q.p = 0$. Therefore, $Jq3$ and $Iq4$ imply that the conjunction $q \text{ in } \{14 \dots\} \wedge p \in nbh.q$ holds infinitely often.

When process q leaves $\{14 \dots\}$ by executing line 16 or **env14**, it decrements $vf2(q, p)$ because of $Mq1$; it therefore also decrements $vf(q, p)$ (even if $vfl(q, p)$ increases). As $vf(q, p)$ is eventually constant, it follows that eventually q remains in $\{14 \dots\}$ and therefore p remains in $nbh.q$. This proves (13).

For the second case, we claim:

$$(14) \quad p < q \Rightarrow Wf(p) \cap \Diamond \Box [q \in need.p] \subseteq \Diamond \Box [q \text{ in } \{15 \dots\} \wedge p \in nbh.q] .$$

This is proved as follows. The fact that q remains in $need.p$ together with $Iq6$ and formula (1) are used to prove that eventually always $req.p.q = 0$, and $gra.q.p = 0$, and $gra.p.q = 0$. This implies eventually always $p \in prom.q$ by $Jq1$, and $p \notin away.q$ by $Jq2$, $Iq3$, and $Iq4$. Yet, the alternative **prom** is not taken anymore. Therefore, weak fairness implies that $q \text{ in } \{15 \dots\} \wedge p \in nbh.q$ holds infinitely often. Again using that $vf(q, p)$ is eventually constant, we get that process q eventually remains in $\{15 \dots\}$ and that p remains in $nbh.q$. This proves (14).

As $q \in need.p$ implies $q \in nbh.p$ by $Iq6$, the formulas (12), (13), (14) combine to yield

$$(15) \quad Wf(p) \cap \Diamond \Box [p \text{ at } 14] \subseteq \bigcup_q \Diamond \Box [p \bowtie q] .$$

With regard to starvation freedom, we use the formulas (12), (13), (14) to prove

$$(16) \quad WF \cap \Diamond \Box [r \text{ at } 14] = \emptyset .$$

This is done by contradiction. Assume that xs is an element of the lefthand expression. Let p be the lowest process with $xs \in \Diamond \Box [p \text{ at } 14]$. Formula (12) gives us a process $q \neq p$ with $xs \in \Diamond \Box [q \in need.p]$. By (5), we have

$$(17) \quad xs \in toIdle(15, q) .$$

If $q < p$, formula (13) gives $xs \in \Diamond \Box [q \text{ in } \{14 \dots\}]$. Together with (17), this implies $xs \in \Diamond \Box [q \text{ at } 14]$, contradicting the minimality of p . If $p < q$, formula (14) gives $xs \in \Diamond \Box [q \text{ in } \{15 \dots\}]$, which contradicts (17). This concludes the proof of (16).

4.9 End of proofs

Theorem 2 follows from the formulas (2) and (5) by repeated application of (4) with (6), (11), (16).

Similarly, Theorem 3 follows from the formulas (2) and (5) by repeated application of (4) with (6), (10), (15).

5 Sketch of the PVS verification

The reader who is familiar with PVS can obtain the dump of the proof script from our website [7]. The proof consists of 183 lemmas that can be verified within three minutes on an ordinary laptop.

For the reader who is not familiar with proof assistants, we give some indications here how we used the proof assistant PVS.

The first thing to do is to declare the state space of the algorithm. This is done by means of the declarations:

```

Process: TYPE FROM nat

state: TYPE = [#
  fork: [Process -> [Process -> int]],
  req, gra, notify, withdraw, ack: [Process -> [Process -> nat]],
  pc: [Process -> nat],
  nbh, need, prom, away, wack, before, prio, after:
    [Process -> finite_set[Process]]
#]
```

The first line says that **Process** is an unspecified subset of \mathbb{N} . The second line declares **state** as a record with the program variables as fields.

In order to inform PVS of the types of the free variables that we are going to use, we declare

```
p, q, r: VAR Process
x, y: VAR state
```

The next point is to define the step relation of the transition system according to Figures 1, 2, 3. We construct it as the union of separate relations for each of the 16 alternatives. As an example, the state modification at line 12 of **forward** is given by

```
next12(p, x): state =
  x WITH [
    'notify(p) := LAMBDA q: x'notify(p)(q) + b2n(x'nbh(p)(q)),
    'prio(p) := {q | x'nbh(p)(q) AND x'before(p)(q)
                  AND NOT x'after(p)(q)},
    'pc(p) := 13
  ]
```

Here $x'nbh(p)(q)$ means $q \in nbh.p$ in state x , and **b2n** is the function that converts a Boolean to a bit. The corresponding step relation is:

```
step12(p, x, y): bool =
  x'pc(p) = 12 AND y = next12(p, x) AND empty?(x'wack(p))
```

When the step relation has been constructed, we turn to the construction and the verification of the invariants. This is a major effort. One of the simpler cases is *Iq8*.

```
iq8(q, r, x): bool =
  x'prom(r)(q) IMPLIES q < r

iq8_rest: LEMMA
  iq8(q, r, x) AND step(p, x, y)
  IMPLIES iq8(q, r, y) OR stepReq(p, x, y)

iq8_Req: LEMMA
  iq8(q, r, x) AND stepReq(p, x, y) AND iq9(q, r, x)
  IMPLIES iq8(q, r, y)

iq8_step: LEMMA
  iq8(q, r, x) AND step(p, x, y) AND iq9(q, r, x)
  IMPLIES iq8(q, r, y)
```

This shows that preservation of *Iq8* only needs *Iq9* at the alternative **req**. When all invariants separately have been constructed, we form the conjunction **globinv** of the universal quantifications of them and prove that **globinv** is inductive:

```
globinv_step: LEMMA
  globinv(x) AND step(p, x, y)
  IMPLIES globinv(y)

globinv_start: LEMMA
  start(x) IMPLIES globinv(x)
```

We formalize state sequences, executions, and weak fairness just as described in Section 4. For instance, function *wf* of Section 4.3 is given by

```

weakly_fair(rel)(xs): bool =
  box(diamond(sem1(disabled(rel))))(xs)
  OR box(diamond(sem2(rel)))(xs)

```

Finally, Theorem 2 is given by

```

liveness: THEOREM % starvation freedom
  weakly_fair_all(xs) AND execution(xs)
  IMPLIES box(diamond(sem1(at(11, p))))(xs)

```

6 Conclusions

The acyclicity invariant introduced by Chandy and Misra for their drinking philosophers [3], was not necessary for liveness, and it was problematic for the message complexity and memory requirements. In our solution, we abandon the acyclicity invariant. We also break the symmetry, in two ways. Firstly by giving higher priority to the lower processes. Secondly by using an asymmetric default fork distribution, in which the higher processes are better off in the sense that they need not request forks.

Originally, we had a weaker outer protocol that only added starvation freedom to the inner protocol, and the inner protocol used the low default fork distribution with $fork.q.r = |q < r|$. When we had completed its proof, we disliked the order in which processes could enter CS. We therefore investigated the high default, and postulated the FCFS property. It came as a surprise to us that this resulted in a somewhat simpler algorithm.

The design of the algorithm was only possible, because we could use the proof assistant PVS [14] to verify the invariants and the progress requirements. We could fruitfully reuse parts of the PVS-proof of the earlier algorithms in the proof of the final algorithm.

We hope that the algorithm or variations of it can be used in the design of practical resource allocation algorithms. Indeed, when the processes compete for several resources, Chandy and Misra [3] suggest to use coloured bottles (forks) with a colour for every resource. This can also be done in our algorithm. There should, however, be some relationship between the resources in the application and the neighbourhoods in our model, and this relationship should be exploited for better performance in the presence of many processes. For applications on the internet, one would need to extend the algorithm with fault tolerance. These extensions are matters for future research.

References

1. M. Abadi and L. Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82:253–284, 1991.
2. B. Awerbuch and M. Saks. A dining philosophers algorithm with polynomial response time. In *Proceedings of the 31st IEEE Symposium on Foundations of Computer Science*, pages 65–74, St. Louis, 1990.
3. K.M. Chandy and J. Misra. The drinking philosophers problem. *ACM Trans. Program. Lang. Syst.*, 6(4):623–646, 1984.
4. K.M. Chandy and J. Misra. *Parallel Program Design, A Foundation*. Addison–Wesley, 1988.
5. E.W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Inf.*, 1:115–138, 1971.
6. E.W. Dijkstra. A strong P/V-implementation of conditional critical regions. Tech. Rept., Tech. Univ. Eindhoven, EWD 651, see www.cs.utexas.edu/users/EWD, 1977.

7. W.H. Hesselink. Partial mutual exclusion.
<http://www.cs.rug.nl/~wim/mechver/partialmx.html>, 2011.
8. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
9. G.J. Holzmann. *The SPIN Model Checker, primer and reference manual*. Addison-Wesley, 2004.
10. L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17:453–455, 1974.
11. L. Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16:872–923, 1994.
12. E.A. Lycklama and V. Hadzilacos. A first-come-first-served mutual-exclusion algorithm with small communication variables. *ACM Trans. Program. Lang. Syst.*, 13:558–576, 1991.
13. N.A. Lynch. *Distributed Algorithms*. Morgan Kaufman, San Francisco, 1996.
14. S. Owre, N. Shankar, J.M. Rushby, and D.W.J. Stringer-Calvert. *PVS Version 2.4, System Guide, Prover Guide, PVS Language Reference*, 2001.
<http://pvs.csl.sri.com>
15. I. Rhee. A modular algorithm for resource allocation. *Distr. Comput.*, 11:157–168, 1998.
16. B.K. Szymanski. A simple solution to Lamport's concurrent programming problem with linear wait. In *Proceedings of the 2nd International Conference on Supercomputing Systems*, pages 621–626. ACM, 1988.
17. J.L. Welch and N.A. Lynch. A modular drinking philosophers algorithm. *Distr. Comput.*, 6:233–244, 1993.